Component-Based Software Development

Todd Urbatsch, CCS-4

collaborators: Tom Evans, Rob Lowrie, Doug Kothe

Talk to Code Strategy Implementation Team

13 August 2002

LA-UR-03-3937

- Define "components"
- Example of high-level components
- Role of components in software development
- Role of components in verification and validation
- Experiences and thoughts
- Demonstration
- Proposals for moving ahead

Goal of Component-Based Software Development

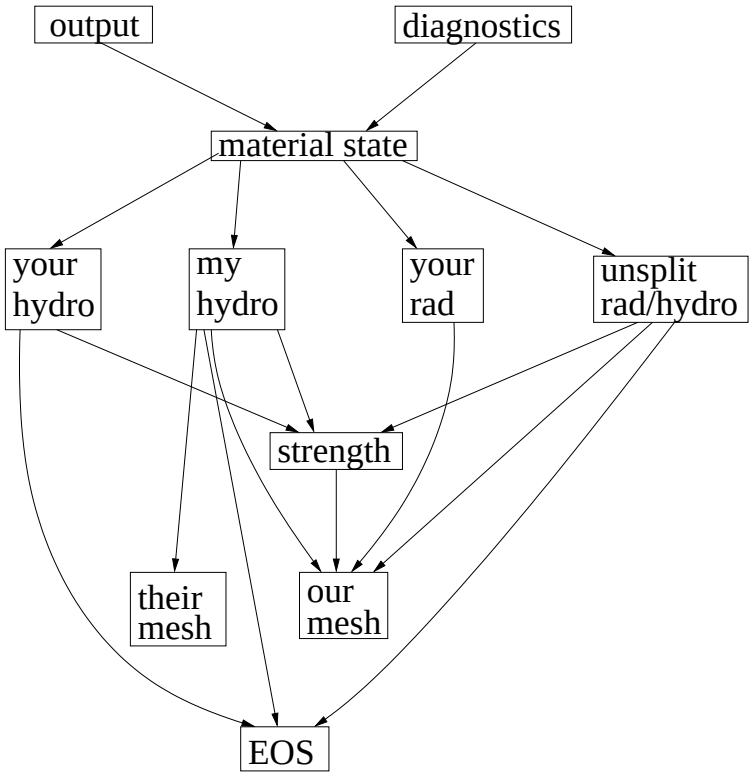
develop high-quality software that we can verify and contribute to validation efforts.

Definition of Components

Components are physically independent software modules designed with standard, clearly defined interfaces that protect them from software changes outside their boundaries. Applications are then composed of components, perhaps even at run time. Because component communication occurs only through well-defined interfaces, an application is changed by modifying one or more components, without fear of disturbing other application components.

- Described by contracts.
  - input
  - function
  - output
- behavior is repeatable

Simple example of high-level components



Each box is made up of lower-level components, possibly shared:

Particle<MeshType>: transport
(simple_test, gray_transport_test, mg_transport_test)

RZWedgeMesh:get_db (simple_one_cell_RZWedge, tstAMR)

Role of Components in Software Development

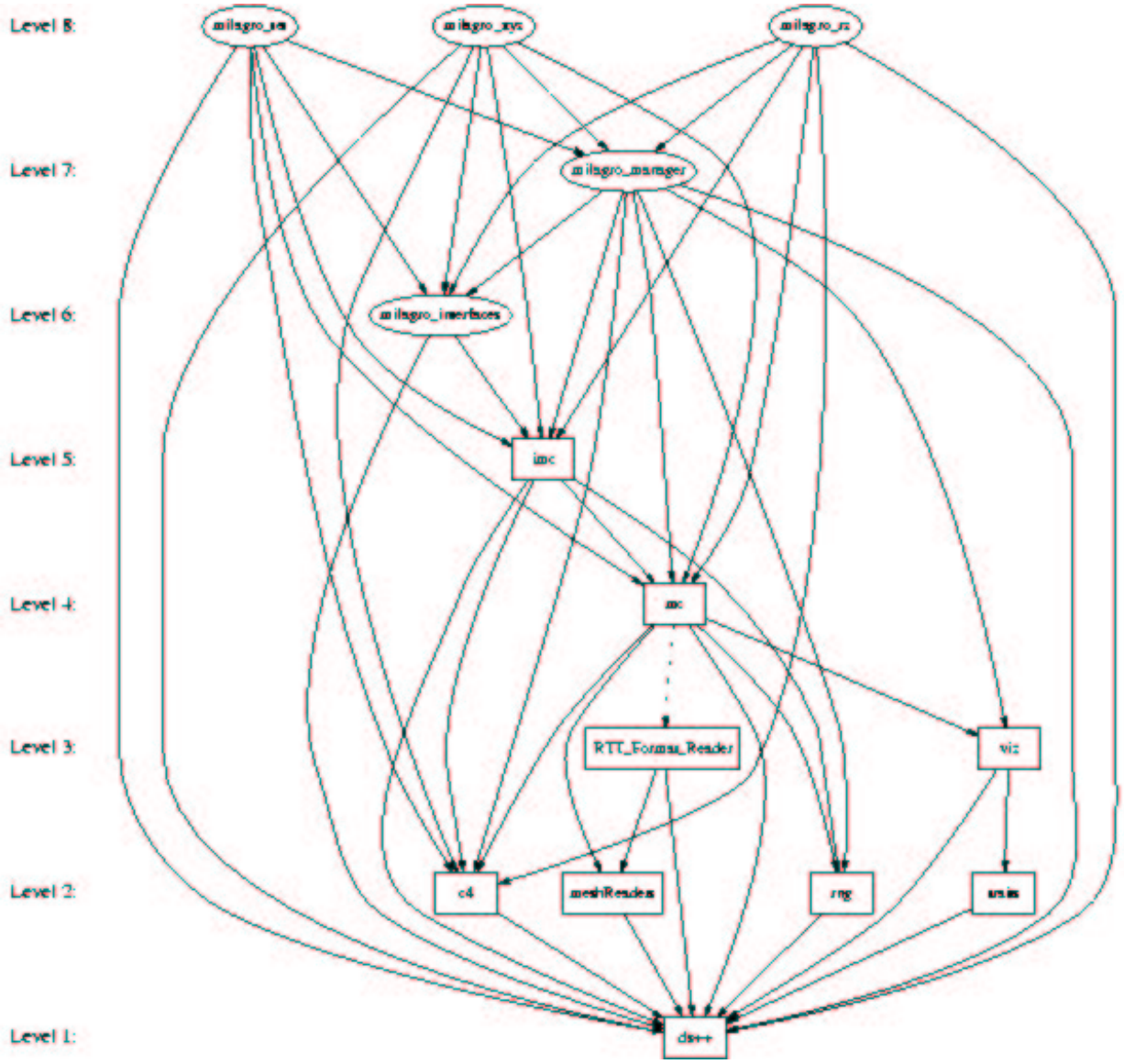
Four aspects of software development:

- Design
- Development
- Archiving
- Principles

Software Development: Design

- components
- levelized design
- Design-by-Contract: Require, Check, Ensure, Insist
- template parameters in generic components
  - Mesh Type (OS_AMR, RZ_Wedge, Tet)
  - Frequency Type (Gray_Frequency, MG_Frequency)

Milagro Levelized Diagram



Note that all arrows GO DOWNWARD. Also, dotted lines indicate a dependency for component testing, not a package dependency. Ovals are Milagro components; boxes are Draco components.

Software Development: Developement

- version control
- build system
- releases
- peer-review
  - pair programming
  - code walk-throughs
  - code reviews
  - post-mortem reviews
- testing
  - unit
  - integral
  - verification
  - shunt
  - regression

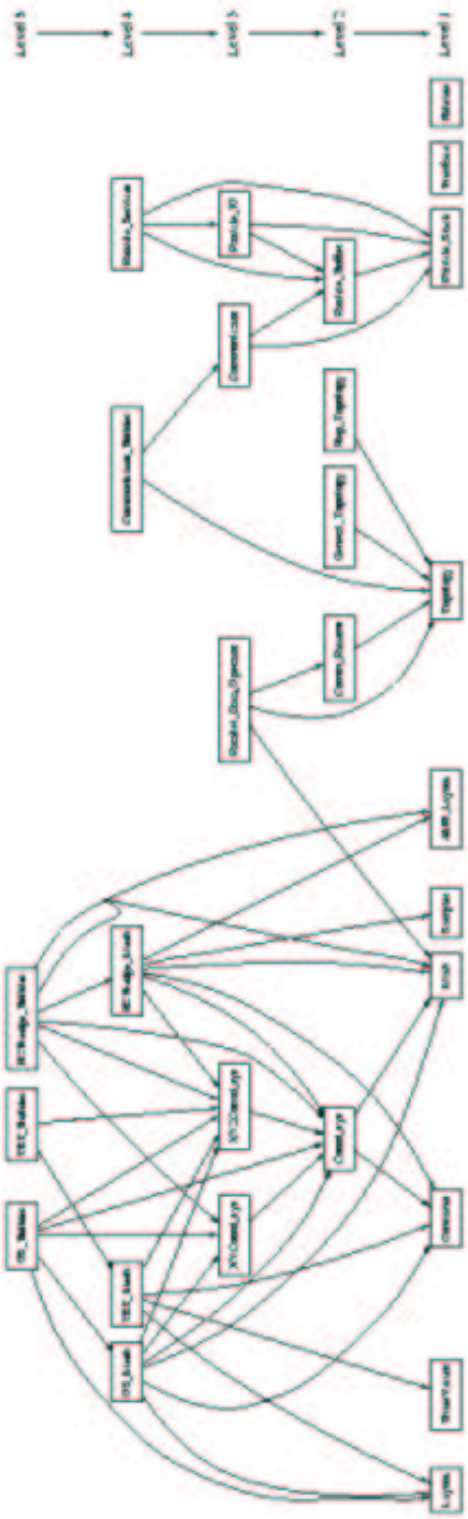
Software Development: Archiving

- requirements documents
- release notes
  - methods manual
  - users' guide
  - verification
- interface documents
- technical memorandums
- research notes

Software Development: Principles

- customer focus/product oriented
- staged delivery/iterative development
- unified process
  - inception
  - elaboration
  - construction
  - transition

Levelization as a Design Analysis Tool



Benefits of Levelized Components

- unit testing is possible
  - isolates bugs
  - effort goes as $2N$ + overhead instead of N^2
- unit tests are repeatable
  - regression
  - component modification
- exposes poor designs
- helps assign responsibility for bugs/issues
- components can be shared or interchanged
  - improvements propagate
  - effort amortized
  - sharing requires overhead - version control mitigates
- does not preclude integral tests

Role of Components in Verification

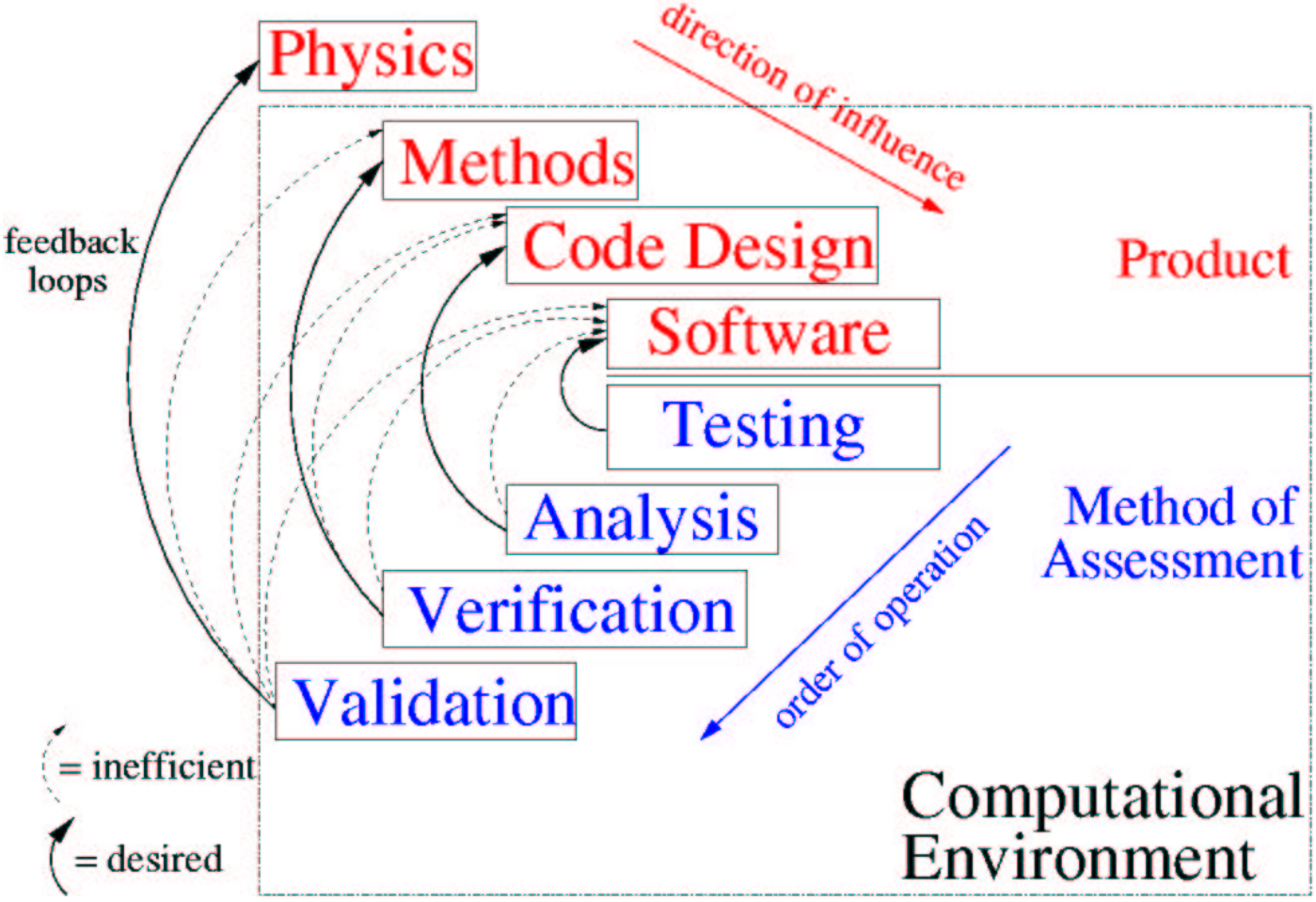
"Are you solving the equations correctly?"

- knowing component is correct
- proving component is correct
- essential for approaching predictive capability
- required for any mathematical expression
  - numerical methods
  - remapping data
  - simply accessing data is an "="
  - etc.

Role of Components in Validation

"Are you solving the correct equations?"

- difficult work
- ongoing
- should not overlap debugging or verification



Conclusions

- proven success with good SQE practices such as components
  - improved quality
  - ability to verify every component
  - code re-use
  - easier maintenance
  - easier to add new capabilities
  - increased confidence
  - one bug in 1M cpu-hours of use
- hope others will adopt some of these practices
- good SQE practices "necessary, but not sufficient"
- good SQE practices don't guarantee success
- make verification and, hence, validation tractable

Future Considerations, 1/3

- Identify and assess existing capabilities
  - + identify the equations
  - + construct levelized designs
    - – conceptual
    - – actual
    - – compare and contrast
  - + identify and prioritize capabilities suitable for conversion to components
- Create a common repository (e.g., sourceforge)
  - + common repository is a Development Environment (DE) in the broad sense
  - + does not dictate components or their interfaces
  - + experts still required to assemble components into codes
- Educate code developers in best SQE practices
- Put forth minimal requirements
  - + version control
  - + releases

Future Considerations, 2/3

- Convert existing high-level capabilities to components
  - + refactor capability into a component
  - + place component in common repository
  - + refactor application code to use the component in the common repository
  - + allows for true unit testing: do it
  - + maintains capability
  - + is anyone willing?
  - + provides for sharing of verified components
  - + allows for interchanging verified components
- Write new low-level components from scratch
  - + can use the best SQE practices from inception of code
  - + should be done anyway
  - + requires cooperation to get used in codes

Future Considerations, 3/3

- Peer review
  - + code sit-downs to assess:
    - – degree of verification
    - – ease of verification
    - – ease of maintenance
    - – ease of use
  - + documents
  - + presentations
  - + Buggy Pageant
- Define consequences for not cooperating
- Do nothing at all; component-based development will eventually take over as it is required.